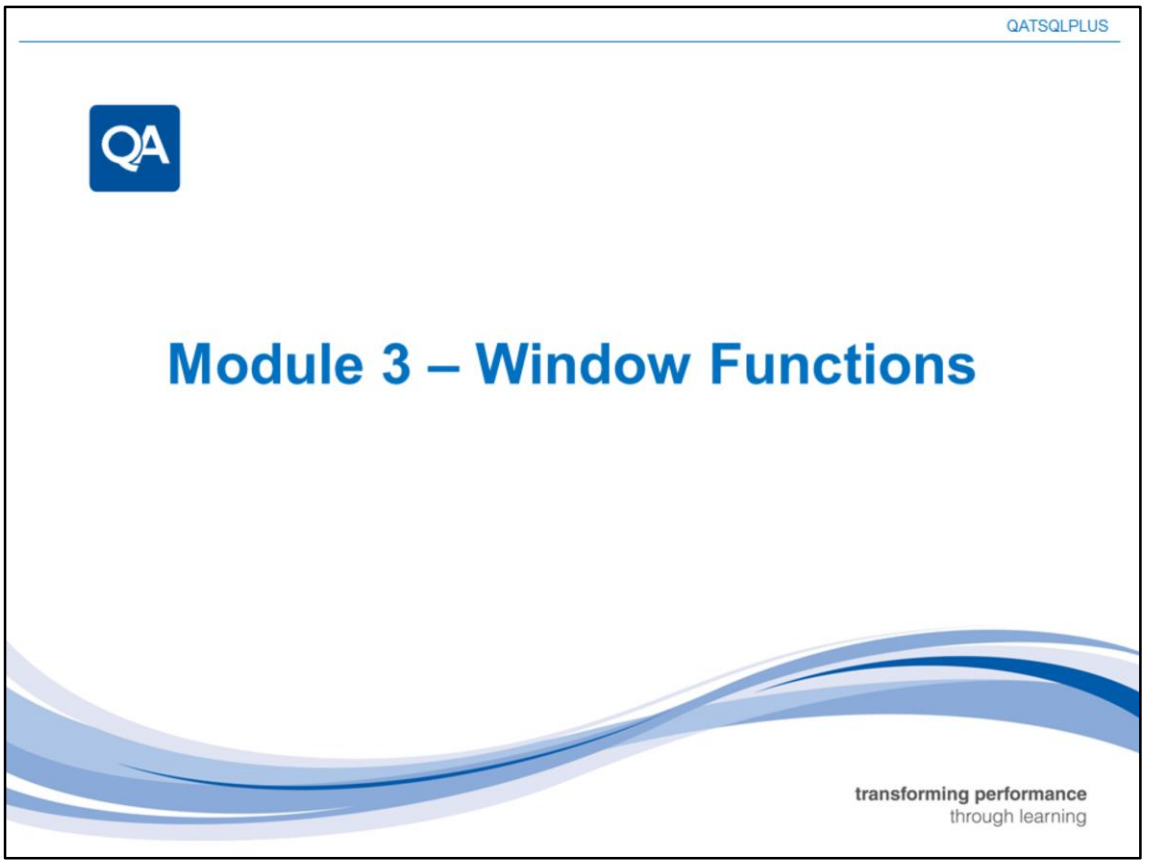
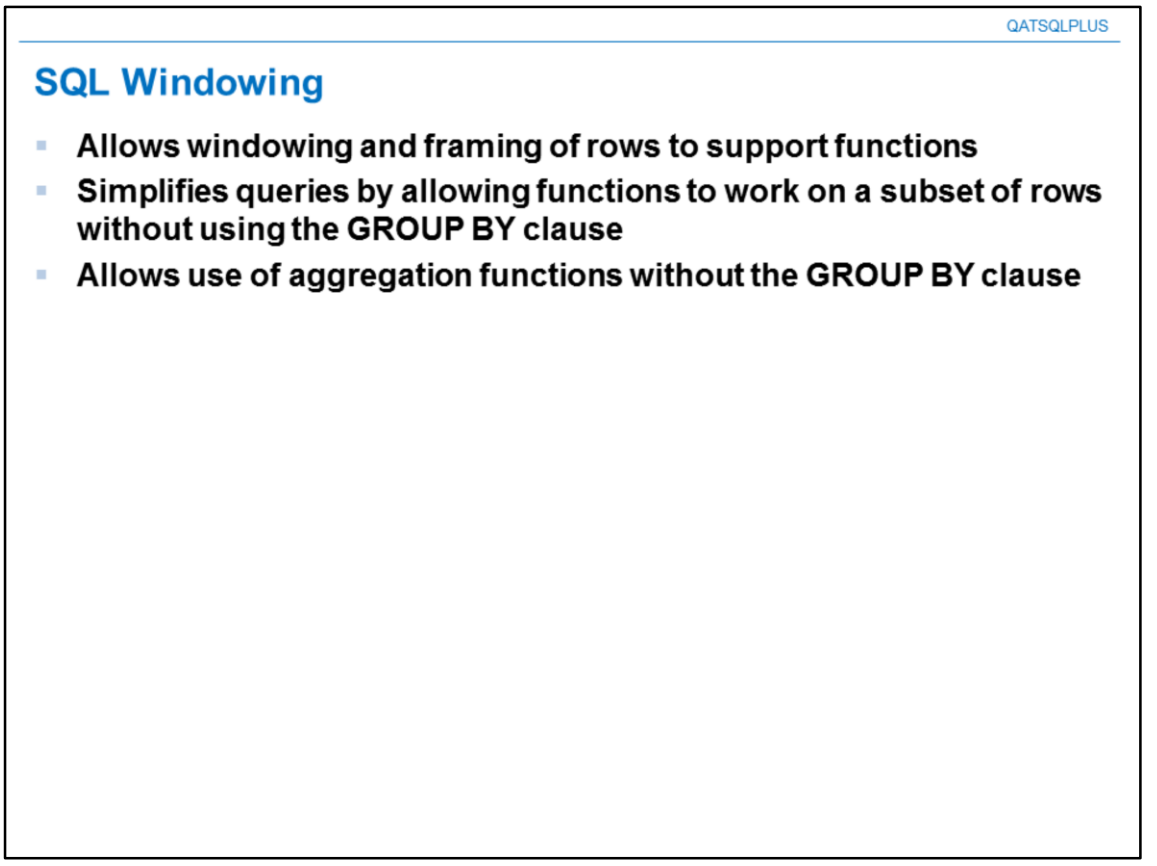

QATSQLPLUS
Module 3 – Window Functions
transforming performance
through learning

QATSQLPLUS

Window Functions

- **SQL window functions**
- **Clauses**
  - Over
  - Partitioning By
  - Rows between
- **Aggregates**
- **Ranking**
- **Positional**
- **Distribution**



When most aggregate queries are executed, a GROUP BY clause is required. This may be an issue if detailed rows of data are required to be shown with the aggregated values. One option to produce this result set, is to use a derived table for the aggregates and then to join this to the original detail rows.

From SQL Server 2005, Window functions have been added which cover multiple groups of functions: aggregates and ranking. Positional and distribution functions have been added in SQL Server 2012.

Example

To produce a percentage contribution of countries to the sales by region, given that the Country Sales view (CountryName, RegionName, Sales).

Using a derived table

```
SELECT C.* , C.Sales / R.RegionSales AS PercentageSales
   FROM dbo.CountrySales AS C
      INNER JOIN
      ( SELECT RegionName, SUM(Sales) AS RegionSales
         FROM dbo.CountrySales
         GROUP BY RegionName) AS R
      ON C.RegionName = R.RegionName
```

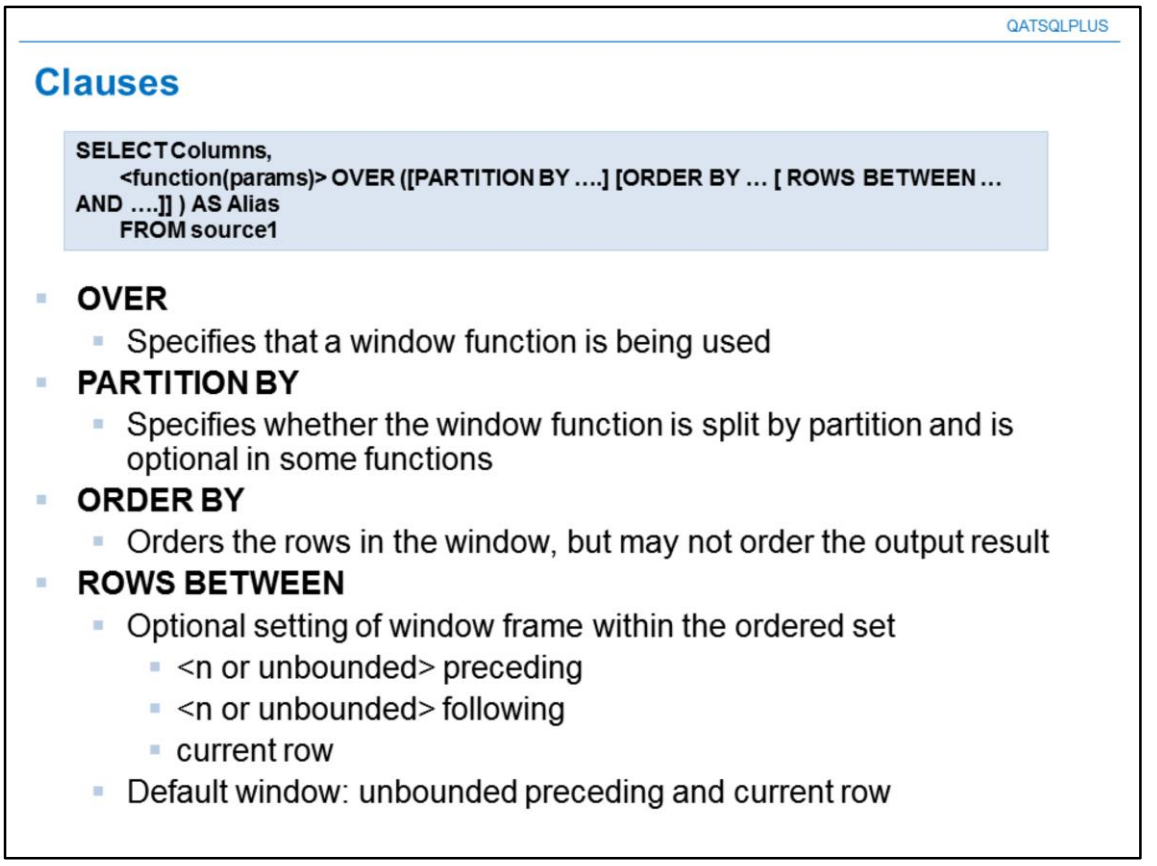
Using a CTE

```
WITH RegionSales AS (
( SELECT RegionName, SUM(Sales) AS RegionSales
        FROM dbo.CountrySales
        GROUP BY RegionName
)

SELECT C.* , C.Sales / R.RegionSales AS PercentageSales
   FROM dbo.CountrySales AS C
      INNER JOIN RegionSales AS R
      ON C.RegionName = R.RegionName
```

Using Window functions

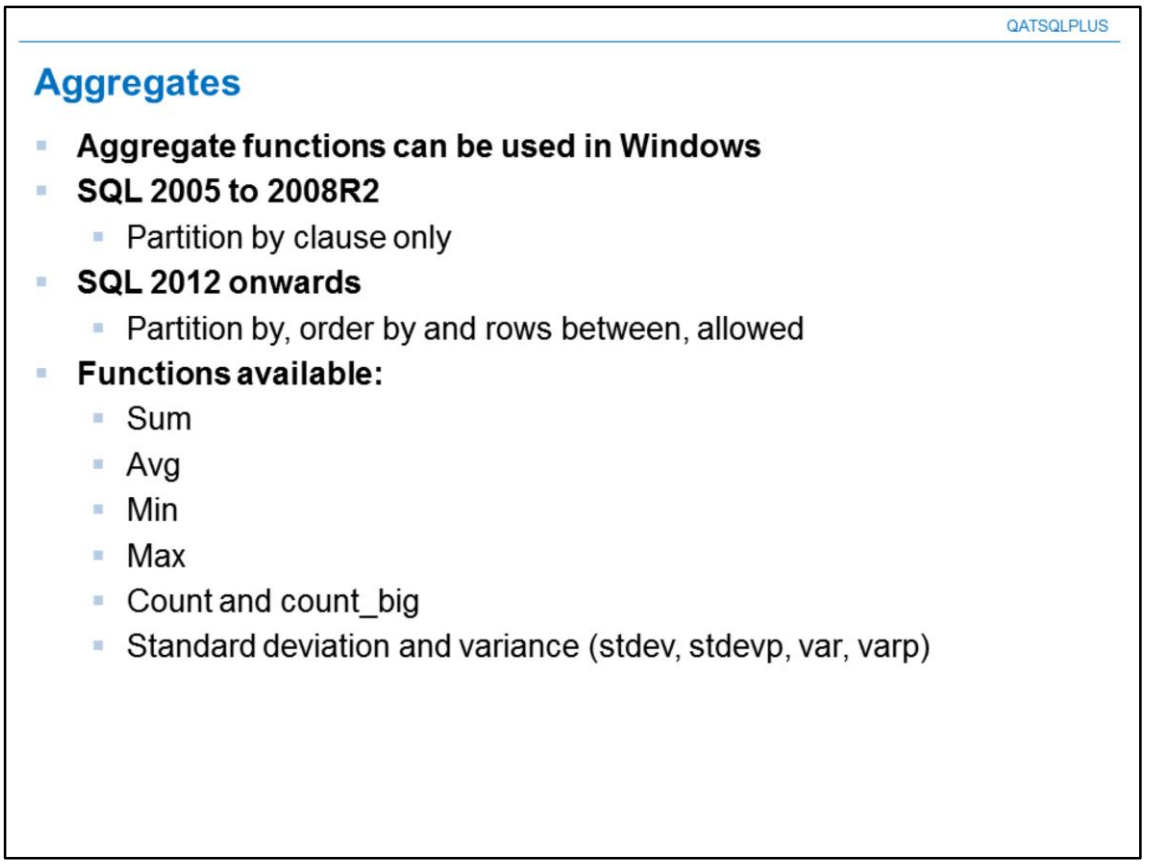
```
SELECT C.* , C.Sales / SUM(Sales) OVER (PARTITION BY RegionName) AS
PercentageSales
   FROM dbo.CountrySales AS C
```



Clauses

- Each window function allows the use of PARTITION BY, ORDER BY and ROWS BETWEEN within the OVER clause
- The PARTITION BY clause is optional
  - If the clause is not used then the complete dataset is used without partitioning
- The ORDER BY clause is optional depending on the function used
  - If the clause is not used then the all rows in the partition will be used for the aggregate function
- The ROWS BETWEEN clause is optional
  - If the clause is used then the ORDER BY clause must also be used
  - If the clause is not specified but the ORDER BY clause is, then the default for the ROWS BETWEEN is unbounded preceding and current row

```
SELECT <columns>,
      function(Column)
        OVER (
          PARTITION BY <columns>
          ORDER BY <columns>
          ROWS BETWEEN start_row_point AND end_row_point
          ) AS Alias
   FROM <query>
```



Aggregates

SQL 2005 to 2008R2:

- Each window aggregate function allowed the use of PARTITION BY – but did not allow for ORDER BY within the OVER clause
- The PARTITION BY clause is optional
    - If the clause is not used then the complete dataset is used without partitioning

SELECT <columns>, SUM(Column) OVER (PARTITION BY <columns>) AS Alias

FROM <query>

SQL 2012:

- Each window aggregate allows the use of PARTITION BY, ORDER BY and ROWS BETWEEN within the OVER clause
- The PARTITION BY clause is optional
    - If the clause is not used then the complete dataset is used without partitioning
- The ORDER BY clause is optional
    - If the clause is not used then the all rows in the partition will be used for the aggregate function
- The ROWS BETWEEN clause is optional
    - If the clause is used then the ORDER BY clause must also be used
    - If the clause is not specified but the ORDER BY clause is, then the default for the ROWS BETWEEN is unbounded preceding and current row

```
SELECT <columns>,
        function(Column)
           OVER (
              PARTITION BY <columns>
              ORDER BY <columns>
              ROWS BETWEEN start_row_point AND end_row_point
              ) AS Alias
    FROM <query>
```

QATSQLPLUS

Aggregates

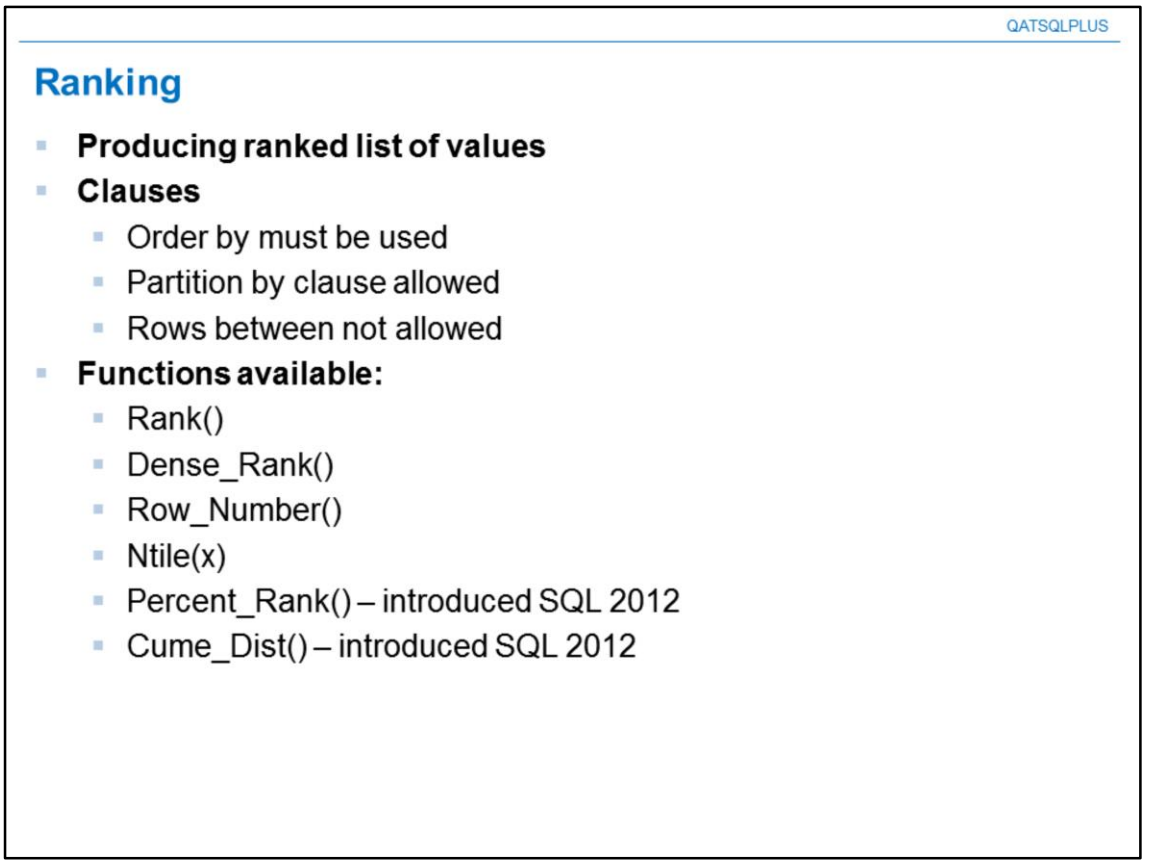
Command outline:

```
SELECT Columns,
    <aggregate function> OVER ([partition by ….] [ORDER BY … [ ROWS BETWEEN … AND ….]] ) AS Alias
    FROM source1
```

Demonstration:

```
SELECT *
    ,sum(TransactionValue) OVER () AS TotalAll
    ,sum(TransactionValue) OVER (PARTITION BY PersonRef) AS TotalPerson
    ,sum(TransactionValue) OVER (PARTITION BY PersonRef ORDER BY PayDT) AS RunningTotalPerson
    ,avg(TransactionValue) OVER (PARTITION BY PersonRef ORDER BY PayDT ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS AvgLast3Person
    FROM account_transactions
    ORDER BY PersonRef, PayDT
```

Person Ref	PayDT	Transaction Value	Total All	Total Person	Running TotalPerson	AvgLast3 Person
3129390	2016-01-02	200.00	800.00	700.00	200.00	200.00
3129390	2016-02-02	200.00	800.00	700.00	400.00	200.00
3129390	2016-03-02	200.00	800.00	700.00	600.00	200.00
3129390	2016-05-02	100.00	800.00	700.00	700.00	166.66
3453543	2016-01-02	300.00	800.00	100.00	300.00	300.00
3453543	2016-01-08	-100.00	800.00	100.00	200.00	100.00
3453543	2016-01-25	400.00	800.00	100.00	600.00	200.00
3453543	2016-02-04	-500.00	800.00	100.00	100.00	-66.66



Ranking

- Each window ranking function allows the use of PARTITION BY and ORDER BY within the OVER clause, but cannot use the ROWS BETWEEEN window frame
- The PARTITION BY clause is optional
    - If the clause is not used then the complete dataset is used without partitioning
- The ORDER BY clause is required

```
SELECT <columns>,
      function(Column)
        OVER (
          PARTITION BY <columns>
          ORDER BY <columns>
      ) AS Alias
   FROM <query>
```

The ranking functions are:

Rank()	Numbers from 1 to n, with duplicate values in the ORDER BY being given the same rank. If duplicates are found the next ranks as skipped (1, 2, 2, 4, 4, 4, 7) where the 2nd 2 would normally be 3rd so the next position is 4th.
Dense_Rank()	Numbers from 1 to n, with duplicate values in the ORDER BY being given the same rank. If duplicates are found the next ranks not skipped and the gaps closed (1, 2, 2, 3, 3, 3, 4).
Row_Number()	Numbers from 1 to n, with duplicate values allowed in the ORDER BY being given the next ranks.
NTile(x)	Splits the data into x blocks as sorted by the ORDER BY clause. If the number of rows is not divisible by x, then the remainder rows are added from the first block. For instance given Ntile(3) with 8 rows, block 1 and 2 would have 3 rows each with block 3 having 2 rows.
Percent_Rank()	Allocates a decimal number from 0.0000 to 1.0000 where the 0 is allocated to the first row and 1 to the last row. Formula is based on the RANK() value: Percent_Rank = (Rows_Rank – 1) / (max(Rank) -1)
Cume_Dist()	Allocates a decimal number up to 1.0000 where 1 is allocated to the last row. Formula is based on the slightly altered RANK() function where equal values are both allocated the higher rank value. Cume_Dist = Rows_Altered_Rank / max(Rows_Altered_Rank)

QATSQLPLUS

Ranking

Command outline:

```
SELECT Columns,
    <ranking function> OVER ([partition by ….] ORDER BY …) AS Alias
    FROM source1
```

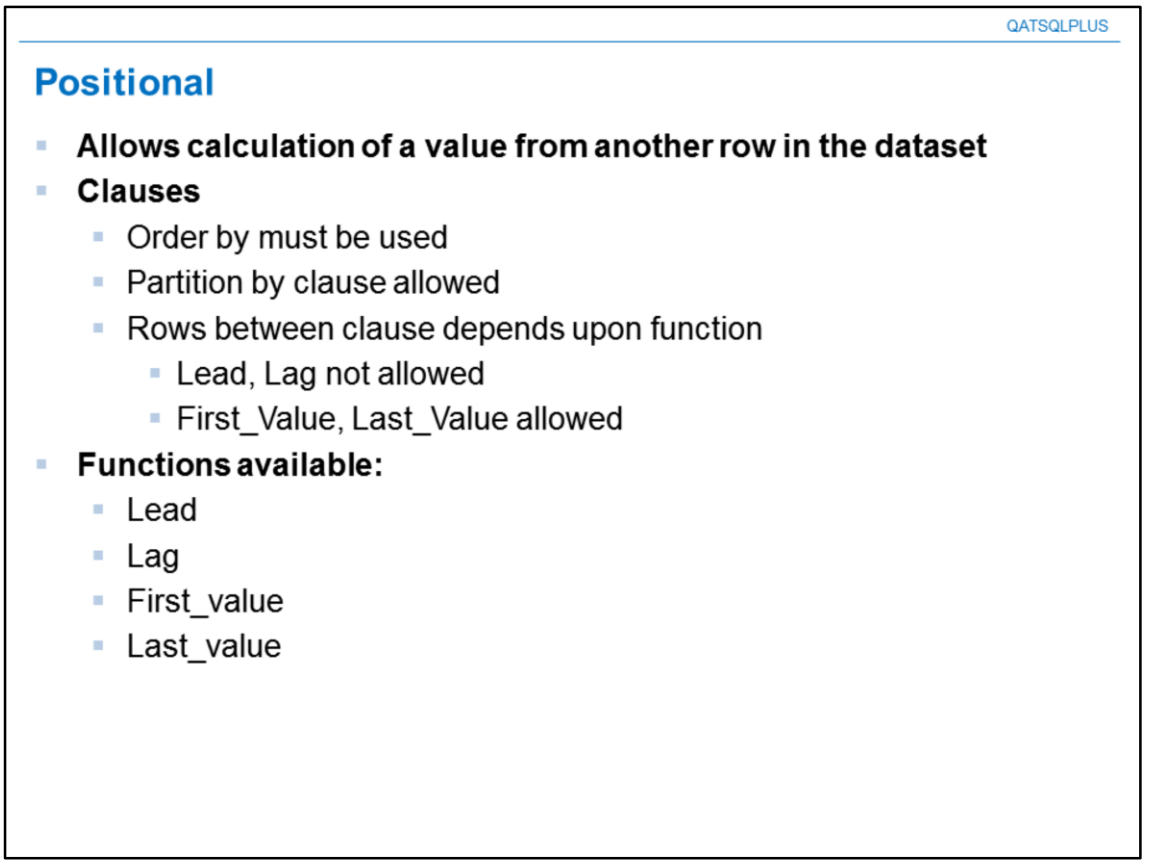
Demonstration:

```
SELECT *
    ,rank() OVER (ORDER BY score DESC) AS Rank
    ,dense_rank() OVER (ORDER BY score DESC) AS Dense_Rank
    ,row_number() OVER (ORDER BY score DESC) AS Row_Number
    ,ntile(3) OVER (ORDER BY score DESC) AS Ntile
    ,convert(decimal(5,4), percent_rank() OVER (ORDER BY score DESC)) AS
Percent_Rank
    ,convert(decimal(5,4), cume_dist() OVER (ORDER BY score DESC)) AS
Cume_Dist
    FROM rank_data
```

	Score	Rank	Dense_ Rank	Row_ Number	Ntile	Percent_ Rank	Cume_Dist
Beta	90.30	1	1	1	1	0.0000	0.1429
Zeta	80.10	2	2	2	1	0.1667	0.4286
Eta	80.10	2	2	3	1	0.1667	0.4286
Alpha	56.70	4	3	4	2	0.5000	0.5714
Gamma	23.50	5	4	5	2	0.6667	0.7143
Delta	12.00	6	5	6	3	0.8333	0.8571
Pi	3.14	7	6	7	3	1.0000	1.0000

Explanation

- **RANK()** – as Zeta and Eta both have a score of 80.10, they are both allocated a rank of 2. The next position is allocated to Alpha as 4.
- **DENSE_RANK()** – as Zeta and Eta both have a score of 80.10, they are both allocated a rank of 2. The next position is allocated to Alpha as 3 as DENSE_RANK() does not skip ranking values.
- **ROW_NUMBER()** – as Zeta and Eta both have a score of 80.10, the system arbitrarily allocates the positions 2 and 3 to these rows.
- **NTILE(3)** – The rows are split into 3 blocks. 7 rows does not divide by 3 evenly, so the extra row is added to the first block.
- **PERCENT_RANK()** – taking the RANK() values and applying the formula (Rows_Rank -1) / (max(Rows) -1), returns the values 0/6, 1/6, 1/6 (as Zeta and Eta have the same rank), 2/6, 3/6, 4/6, 5/6 and 6/6.
- **CUME_DIST()** – taking the RANK() values and applying the formula Rows_Altered_Rank/ max(Rows_Altered_Rank) , returns the values 1/7, 3/7, 3/7 (as Zeta and Eta have the same rank), 3/7, 4/7, 5/7, 6/7 and 7/7.



Positional

Positional window functions were added in SQL Server 2012. This allows for the retrieval of a column value from another row in the same partition.

- Each window positional function allows the use of PARTITION BY and ORDER BY within the OVER clause, and may allow the use the ROWS BETWEEEN window frame
- The PARTITION BY clause is optional
    - If the clause is not used then the complete dataset is used without partitioning
- The ORDER BY clause is required
- The ROWS BETWEEN clause, if allowed, would default to unbounded preceding to current row

```
SELECT <columns>,
      function(Column)
        OVER (
          PARTITION BY <columns>
          ORDER BY <columns>
      ) AS Alias
   FROM <query>
```

The ranking functions are:

- **Lead(column, x, missing_row_value)**
    - Retrieves the column value from the row which precedes the current row, by x rows in the partition (if used) or the complete set
    - If there is no row x rows before the current row then missing_row_value will be returned
    - If missing_row_value is not given, then the default is NULL
    - Lead cannot use the ROWS BETWEEN clause
- **Lag(column, x, missing_row_value)**
    - Retrieves the column value from the row which follows the current row, by x rows in the partition (if used) or the complete set
    - If the there is no row x rows alter the current row then missing_row_value will be returned
    - If missing_row_value is not given, then the default is NULL
    - Lag cannot use the ROWS BETWEEN clause
- **First_Value(column)**
    - Retrieves the column value from the first row within the current partition if a partition is used, or the complete dataset
    - ROWS BETWEEN clause default to unbounded preceding to current row, which should be correct for most applications
- **Last_Value(column)**
    - Retrieves the column value from the last row within the current partition, inside the ROWS BETWEEN window frame, if a partition is used, or the complete dataset
    - ROWS BETWEEN clause defaults to unbounded, preceding to current row, which would normally return the current row
        - In most applications it would be better to specify the window frame for this function

```
SELECT *, LAST_VALUE() OVER (PARTITION BY Column1 ORDER BY Column2
ROWS BETWEEN CURRENT ROW and UNBOUNDED FOLLOWING)
```

QATSQLPLUS

Positional

Command outline:

```
SELECT Columns,
    <positional function>(column, params) OVER ([partition by ….] ORDER BY …. ROWS
BETWEEN …. AND ….) AS Alias
    FROM source1
```

Demonstration:

```
SELECT*
    , lag(PayDT,1,NULL) OVER (PARTITION BY PersonRef ORDER BY PayDT) AS PrevDT
    , lead(PayDT,1,NULL) OVER (PARTITION BY PersonRef ORDER BY PayDT) AS NextDT
    , first_value(PayDT) OVER (PARTITION BY PersonRef ORDER BY PayDT) AS FirstDT
    , last_value(PayDT) OVER (PARTITION BY PersonRef ORDER BY PayDT
        ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS LastDT
    FROM account_transactions
    ORDER BY PersonRef, PayDT
```

Person Ref	PayDT	TransactionValue	PrevDT	NextDT	FirstDT	LastDT
3129390	2016-01-02	200.00	NULL	2016-01-02	2016-01-02	2016-05-02
3129390	2016-02-02	200.00	2016-01-02	2016-03-02	2016-01-02	2016-05-02
3129390	2016-03-02	200.00	2016-02-02	2016-05-02	2016-01-02	2016-05-02
3129390	2016-05-02	100.00	2016-03-02	NULL	2016-01-02	2016-05-02
3453543	2016-01-02	300.00	NULL	2016-01-08	2016-01-02	2016-02-04
3453543	2016-01-08	-100.00	2016-01-02	2016-01-25	2016-01-02	2016-02-04
3453543	2016-01-25	400.00	2016-01-08	2016-02-04	2016-01-02	2016-02-04
3453543	2016-02-04	-500.00	2016-01-25	NULL	2016-01-02	2016-02-04

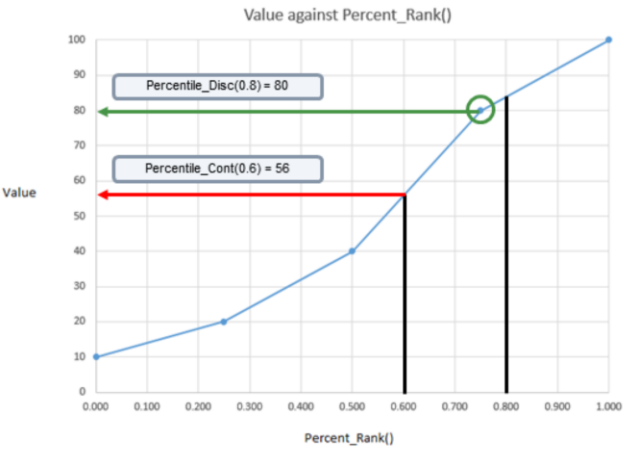
Explanation

- The example shows the dates of the previous, next, first and last payments made
- Each is partitioned by the PersonRef and ordered by the PayDT
- The lag, lead and first_value use the default ROWS BETWEEN window frame (unbounded preceding and current row)
- The last_value needs to have the window frame set to current row and unbounded following

QATSQLPLUS

Distribution

- **Returns a value from the ranked / ordered list matching the percentile given**
- **Clauses**
  - Order by must be used
  - Partition by clause allowed
  - Rows between not allow
- **Functions available:**
  - Percentile_Cont
  - Percentile_Disc



Distribution

- Each window ranking function allows the use of PARTITION BY and ORDER BY within the OVER clause, but cannot use the ROWS BETWEEEN window frame
- The distribution function use an additional WITHIN GROUP clause
- The PARTITION BY clause is optional
  - If the clause is not used then the complete dataset is used without partitioning
- The ORDER BY clause is placed in the WITHIN GROUP clause and is required
- The ROWS BETWEEN window frame is not allowed

```
SELECT <columns>,
    function(value)
      WITHIN GROUP (ORDER BY <columns>)
      OVER (PARTITION BY <columns>) AS Alias
  FROM <query>
```

Examples below use the following values: 10, 20, 40 , 80, 100

Value	Percent_Rank()
10	0.000
20	0.250
40	0.500
80	0.750
100	1.000

The ranking functions are:

Percentile_Cont(value)

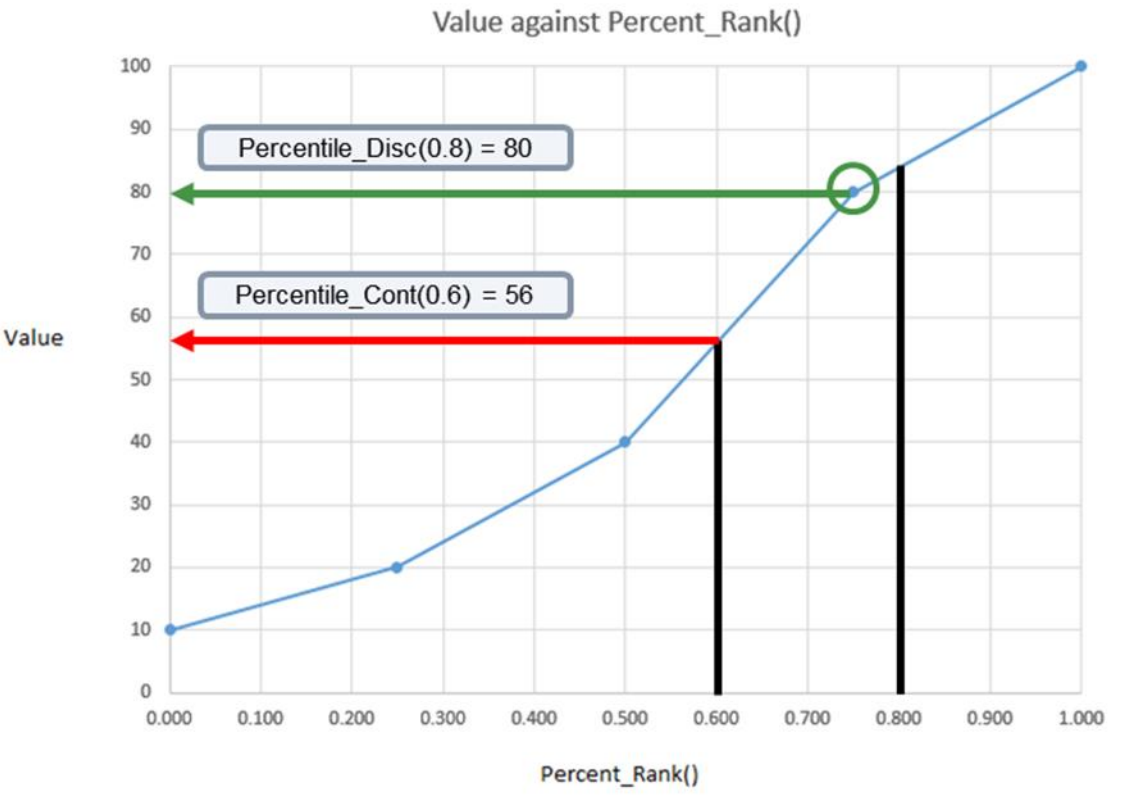
- Returns the interpolated value that would be on a line graph for percent_rank = value.
- The returned value could be a value that does not exist in the input set.
- Given the example data above the result would be 56.

```
SELECT DISTINCT Percentile_Cont(0.6) WITHIN GROUP (ORDER BY Value)
OVER () AS PC
    FROM DemoValues
```

Percentile_Disc(value)

- Returns the value lower than the percent_rank given.
- The returned value would be a value from the input set.
- Given the example data above the result would be 80. 0.8 is not a percent_rank() that matches an exact value so Percentile_Disc takes the next lower value.

```
SELECT DISTINCT Percentile_Disc(0.8) WITHIN GROUP (ORDER BY Value)
OVER () AS PC
    FROM DemoValues
```



QATSQLPLUS

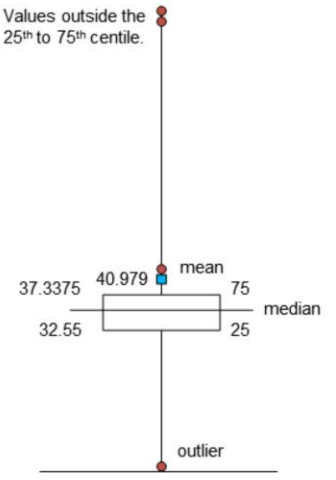
Distribution

Command outline:

```
SELECT Columns,
     <distribution function>(param) WITHIN GROUP (ORDER BY .... )AS Alias
     FROM source1
```

Demonstration:

```
SELECT DISTINCT
     AVG(ResultValue) OVER () AS Mean,
     PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY ResultValue
ASC) OVER () AS Centile25,
     PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY ResultValue
ASC) OVER () AS Median,
     PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY ResultValue
ASC) OVER () AS Centile75,
     PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY ResultValue
ASC) OVER () AS  NextLowestToMedian
     FROM Results
```



Results data:
0.903, 23.45, 30.00, 32.31, 33.27, 34.21, 34.50, 36.00, 36.81, 37.00, 37.45, 42.903, 95.90, 99.00

Mean	Centile25	Median	Centile75	NextLowestTo Median
40.979	32.55	35.25	37.3375	34.5

Explanation

Taking a list of values as shown, the mean can be calculated easily.

Statisticians may like to know what the 25th and 75th centile are for the set. For this use the PERCENTILE_CONT(0.25) and PERCENTILE(0.75) which would give those values.

For the MEDIAN (middle value) the PERCENTILE_DISC(0.5) could be used but this will round down to the next value. In this case the MEDIAN value is 34.5, although 34.5 has a percent_rank() value of 0.4615.

QATSQLPLUS

Exercise